By Carl Malamud

Streams Helps Comm-based Services Go with the Flow

As Part of AT&T's Unix System V, Streams Delivers a Standard Set of Communications Interfaces and Programming Tools that Gives Network Programmers a Modular Approach to Complex, Layered Protocols

In the Dark Ages of data communications, networks were just another program, adjunct to a general-purpose operating system, like a word processor or a graphics routine. But in today's network-intensive environment, that's no longer the case. Network-based applications are on the way to becoming the norm, and the network itself has become another operating system support tool, like file systems, lock managers, and CPU scheduling mechanisms.

With the advent of layered communications protocols, however, the situation has grown decidedly more difficult. The very complexity of these protocols demands a framework that allows them to be implemented in a modular fashion, thus making it possible to build up sophisticated communications services from simpler building blocks. Essentially, a framework of this sort needs to furnish a set of standard interfaces between the modules so that one module need not know the details of its neighbor. Once written, these modules can be used over and over again in different combinations, giving network programmers a highly portable, easily reusable set of resources. For instance, it's possible to arrange the protocol stack so that X.25 (implemented as one or more modules) can be located simultaneously under the network layers of both TCP/IP and OSI.

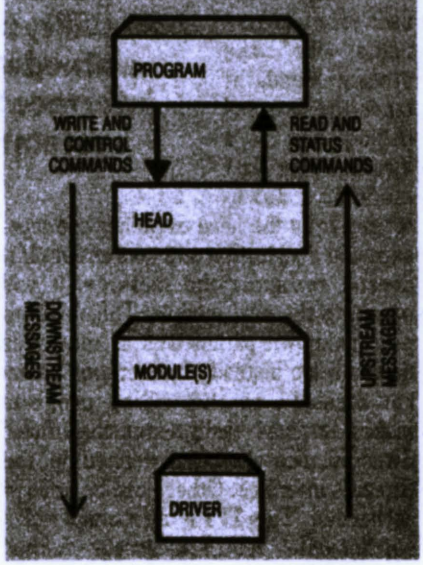
In the Unix world, support for network operations comes through Streams, which is included as a standard part of AT&T's Unix System V. Streams itself is not a network component, but a framework that makes it easier to implement communications-based services.

Streams is not the only such mechanism, of course. But its presence in Unix makes it a factor in many key products: Sun Microsystems Inc.'s and AT&T's Unix, Digital Equipment Corp.'s Ultrix, and IBM's AIX, for example. Other vendors, such as Novell, also are adopting Streams for their operating systems.

In its broadest sense, Streams is a standard set of communications interfaces and programming tools that reside in the operating system kernel. It provides a way of passing messages first between an application program and a network protocol stack and finally to a network interface card.

Figure 1: Pieces of a Stream

A basic stream comprises a head, which establishes the interface to the rest of the system, and a driver. Modules are added to a stream by using a POP system call.



One of the prime advantages to a mechanism like Streams is that it ensures portability by establishing a standard program interface to operating system resources. Moving a Streams-based application to another system can be as simple as recompiling it (and for binary compatibility, it's not even necessary to recompile). True, this ideal is not always realized, but it is the goal.

The other advantage is interoperability, easily seen in the interaction between an application and the network. Many applications directly invoke the services furnished at the network transport layer, providing reliable, end-to-end communications. Streams supplies a way for an application to submit messages to a transport module. If network designers write their modules so that all accept the same types of messages, the application program is able to choose the appropriate transport stack for the job. The same message (for example, "MAKE CONNECTION") can establish a connection on TCP/IP, OSI's TP, and Novell's SPX transport modules.

Coupled with Streams are sets of standard message definitions. In the case of the transport layer, the Transport Provider Interface (TPI), also a part of Unix System V, defines the messages that a transport module can accept and act upon. Thus, any TPI-compliant transport

Streams

module can service such messages, thereby establishing a protocol-independent interface to the application.

Streams provides the framework for interoperability; standard interfaces make that framework a reality. Network programmers—the people who develop databases and scheduling and conferencing systems—can thus concentrate on their work and not worry about the mechanics of the lower layers of the network.

The Streams specification comprises a collection of system calls, function calls, kernel resources, and kernel utility routines. The primary facilities that it offers are buffer management, flow control, scheduling, multiplexing, asynchronous operation, and error and trace logging.

Every individual stream consists of a stream head and a driver, between which modules can be placed. The driver is the end of the stream (see Figure 1). The stream head is a set of routines that establishes the interface to the rest of the system. WRITE and IOCTL (I/O control) requests are turned into messages and sent to the stream. Read requests, in comparison, take data from the stream and pass it to the user. The modules process messages transmitted between a stream head and driver.

The best way to understand Streams is to start with the way I/O processing is done on a local computer. Any device (say, a network interface card) that is accessed by an application has a device driver. Each device driver has a set of commands that signal the type of operation to be performed. Streams comes into the picture by providing the common interface to the network card or to any other device.

Traditional device drivers are based on a character I/O mechanism that lets users read or write one character at a time. Streams operates in the same manner, but it also allows users to dispatch an entire message to the device driver for processing. The message is placed in a queue until the driver is ready to process it. This message I/O feature is one of the facets of Streams that makes it particularly well suited for the network environment, where a shared medium makes messages more efficient than sending single characters.

As noted, a stream consists of a head, modules, and driver. The head contains a set of routines that communicate between the operating system and the application and the rest of the stream. The stream head and driver are connected automatically when the device is opened. In other words, a stream is opened just as a file would be. When the node representing the device on the file system is opened, the device driver is connected to the stream head. Modules can then be added to the stream. Standard operating system calls can be used to open and close the stream.

In addition to standard I/O operations such as READ and WRITE, the stream head supports additional commands. Two important ones are PUSH and POP, which add and remove modules from a stream. When the stream is opened for the first time, it actually consists of a head and driver. The user program can then PUSH modules onto the stack, in between the stream head and the driver. Additional modules are PUSHed between the stream head and the highest-level module.

To add a module onto the stack, the programmer issues an IOCTL call, giving the PUSH argument and giving the address of a module that will be added to the stream. POP would be used to delete a module. PUSH and POP are examples or arguments that the stream head deals with directly. But if IOCTL has an unknown argument, the stream head will generate a message and send it downstream.

When the user sends a command to the stream head, it takes the data and packages it as a message. The first module downstream from the head examines the message and decides if it will take action. If it doesn't recognize the message, it will pass it to the next module. The ability to pass unknown parameters as messages downstream instead of generating an error message to the calling program makes Streams easily extensible. Basic Streams messages are defined in the Streams interface. Others can be defined as needed. The TPI is an example of a supplemental set of messages defined for a particular set of modules and drivers.

Streams ensures portability by establishing a standard program interface to operating system resources.

Streams' message-handling capabilities also provide a more advanced approach to I/O processing. A program can issue a PUT function call, which alerts the stream head that data from some location in memory is to be sent downstream. The head gets the message from user (application) memory, which consists of two parts, control and data.

The control part of the message indicates what kind of message is being sent. The data part is what the interpreter of the message will use. Normally, the program generates protocol messages that are interpreted by modules on the stream, modified, then sent along to the next module or driver.

For example, the top level of the stream might be a transport layer that supplies service primitives to send and receive data. When the transport layer receives a message, it would add the transport header, the sequence number of the message, and any flow-control information for the remote node on the network.

The transport module would then send the message to the network module, which would add the network-level header containing routing information. Then, the message would be handed off to the device driver, in this case a network interface card. The data link information would be added to the message, and the message would be sent onto the LAN.

At the remote end, the reverse would occur. An Ethernet driver, for instance, would receive the message datagram and package it into a Streams message, which would hand it up to the network layer. The network-layer module would strip the routing header and hand the message to the transport layer. The transport module would determine the transport layer address, check the sequence of the message, and then hand it back up to the stream head, which would deliver it to the user program.

Each Streams component has a pair of message queues, one for incoming messages, the other for outgoing mes-

Streams

sages (see Figure 2). The queues establish flow control to prevent messages from overloading a module or driver. They also expedite data services.

Of the two message queues, one is for upstream messages; the other is for those headed downstream. The queues are simply areas of memory managed by a module large enough to hold several messages. When a module sends a message downstream, it invokes the PUT procedure of its downstream module. This allows a module to pass messages to the next module by placing it in a queue for processing. The PUT procedure examines the message and decides if it needs immediate servicing. If not, it will PUT the message onto the queue.

The queue itself consists of a series of linked messages. The head of the queue points to the first message, which in turn contains the address of the next message. The last message, or tail of the queue, has a pointer back to the head.

Normally, a message is PUT in the tail of the queue and the pointer is adjusted to point to the queue head. But certain classes of messages (those with high-priority status) are not subject to flow control and are added not to the tail but to the head, behind any existing high-priority messages.

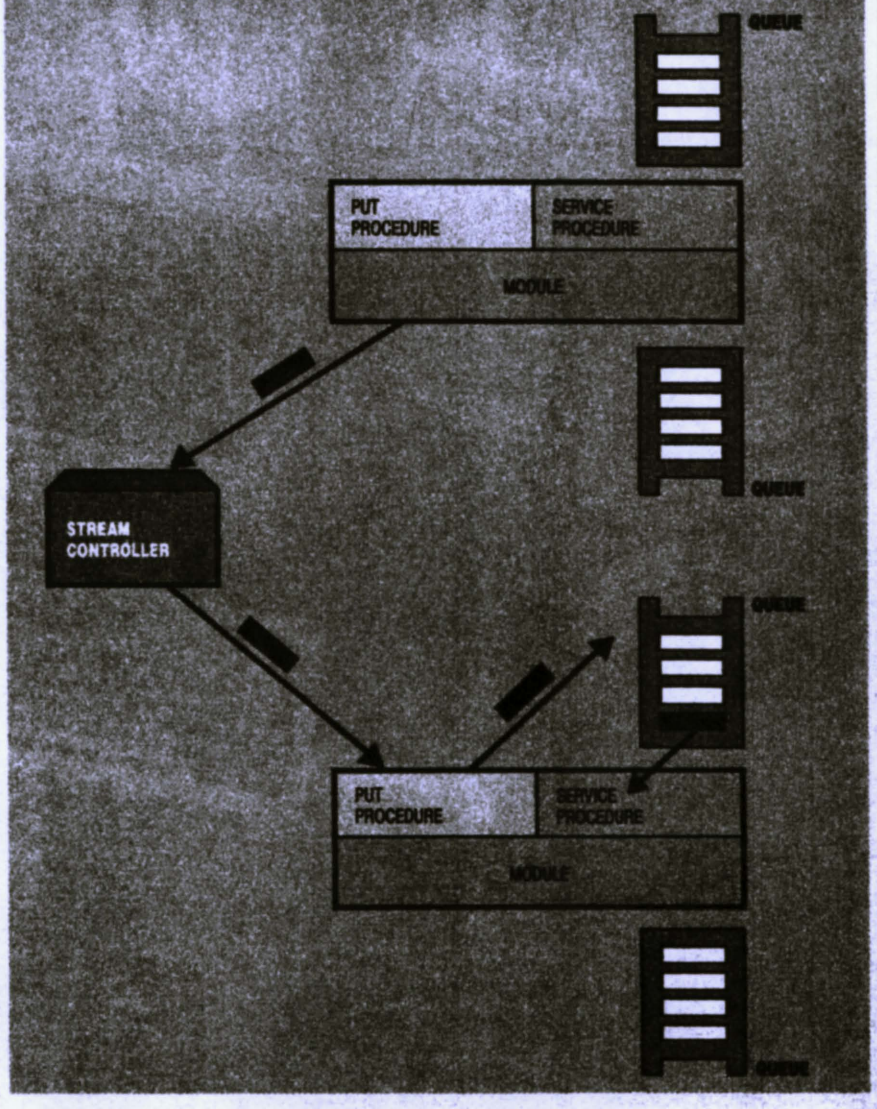
Besides the PUT procedure, a Streams module also has a service procedure associated with it. It is scheduled to execute whenever there is work to be done, and it periodically empties the queue of all messages. At times, a module will be unable to send a message because the downstream module is unable to process it. When this occurs, the stream is blocked. If the downstream queue is blocked, the local module may not be able to empty its own queue, bringing the stream to a halt until the downstream queue is unblocked.

In most computing environments, it is necessary to share drivers among multiple users, which can be accomplished through the Streams multiplexer. A transport-layer protocol such as TCP will support multiple users and is a good candidate for multiplexed Streams.

A multiplexed stream is actually two or more streams stacked atop one another (see Figure 3). The multiplexing module is used to route data to the appropriate stream. Multiplexing can take place in either direction, with several upper streams sharing the multiplexing module or several lower streams under the control of the multiplexing module.

Figure 2: Stream Flow Control

Each component of a stream has two message queues, one for incoming messages and the other for outgoing messages. They ensure flow control to prevent messages from overloading a module or driver and also expedite data services.



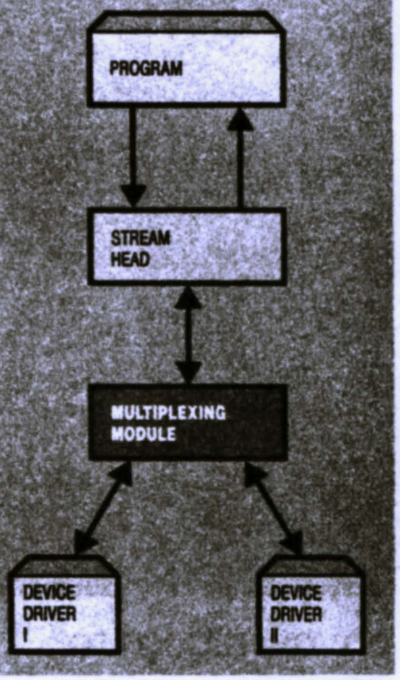
Multiple lower streams are created by first opening two separate streams, one consisting of the stream head and the multiplexing module, and the other, a stream head and driver. Next, the lower stream is connected to the multiplexing driver using an I-LINK command, which instructs the lower module to send all upstream data to the multiplexing module. All data from the multiplexing module is sent to the lower driver. More lower streams can be added using I-LINK commands. It also is possible to add streams on top of this configuration, again using I-LINK.

Although Streams provides a mechanism for passing messages, it only defines those basic messages used to construct a stream, such as PUSH or POP (see table). In a networking environment, additional messages are needed to perform real work. The Transport Provider Interface defines a common class of messages

Streams

Figure 3: Multiplexing Modules in Streams

A multiplexed stream is actually two or more streams stacked atop one another. A multiplexing module routes data to the appropriate stream.



Calling All Streams

FUNCTION	PURPOSE
OPEN	Open a stream
CLOSE	Close a stream
READ	Read data from a stream
WRITE	Write data to a stream
IOCTL	Control a stream
PUSH	Add a module to a stream
POP	Remove a module from a stream
GET	Receive the message at a stream head
PUT	Send a message downstream
POLL	Notify the application program when selected events occur on a stream

that are recognized by different transport layer modules. The Transport Level Interface (TLI) provides a functional interface to the TPI, which is the actual message format understood by the transport protocols. Thus, the TLI library translates a function call request for a transport service into a TPI message to use the service provided by the underlying protocols.

A TPI-compliant transport level module will service any application layer program that uses the TLI interface. Instead of programming directly to Novell's SPX, for example, the programmer writes an application that uses the TLI library calls. At run time, the program can be run using SPX, OSI, or TCP/IP protocol stacks. It's thus possible to change the underlying transport mechanism without rewriting programs. And since TLI has been adopted by so may vendors, the network components of a program have become very portable.

TLI consists of two components: a TLI library and a TLI module (see Figure 4). The library is linked into the application program and becomes a support library much in the same way remote procedure call (RPC) libraries are linked to an application. The TLI library accepts TLI calls in the main body of the program and performs the required operations to deliver a message to the stream head.

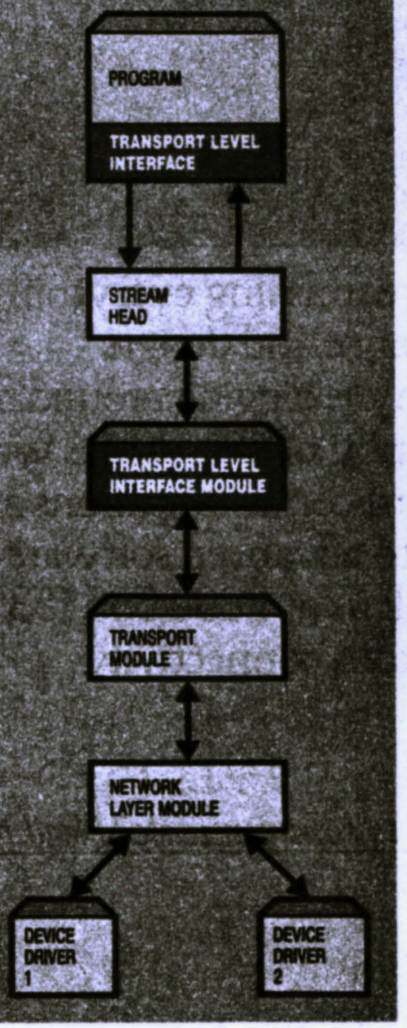
The TLI module is PUSHed onto the stream when it is opened. It then accepts requests from the TLI library and translates some of them into standard TPI messages. The TLI library allows a programmer to develop network-based programs without worrying about the type of transport stack being used. For example, a programmer can issue a T-CONNECT function call to open a connection on any supported transport stack.

The T-CONNECT call is translated into a T-CONN-REQ (transport connection request) message, which is passed down to the transport provider, a module that accepts TPI messages and then provides the services on a particular network. The TLI module then waits until it receives a T-CONN-CON (transport connection confirmed) back from the transport provider. At that point, it returns the information to the stream head, which fills in the address of the remote connection.

A wide variety of TPI messages (and corresponding function calls) have been defined for Streams, including error handling and expedited data, which further extends the usefulness of the Streams programming interface. The basic functionality is the same as if a programmer were writing directly to the transport layer. The advantage of this

Figure 4: Streams Goes to the Library

The Transport Level Interface consists of a TLI library and TLI module. The library links to the application program and accepts TLI calls, delivering messages to the stream head. The TLI module is pushed onto the stream when it is opened. It accepts and translates requests from the TLI Library.



Now there's more than one way to reach NetView.™

With Astrocom's 2300 DSU/CSU the road to LPDA-2 is smooth and straight.

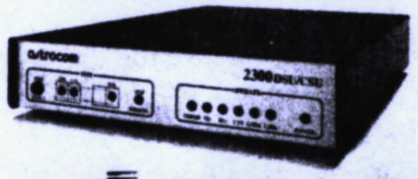
The Astrocom 2300 DSU/CSU with the LPDA-2 option offers everything you need for complete NetView™ compatibility.

All the diagnostic options. The easy DDS access.

The variable speed 2300 DSU/CSU supports speeds of 2.4, 4.8, 9.6, 19.2 and 56K bps. RS232 and V.35 interfaces are co-resident. This combination provides the flexibility to reconfigure your network without changing the DSU/CSU.

Plus, it's attractively priced and backed by responsive, expert service.

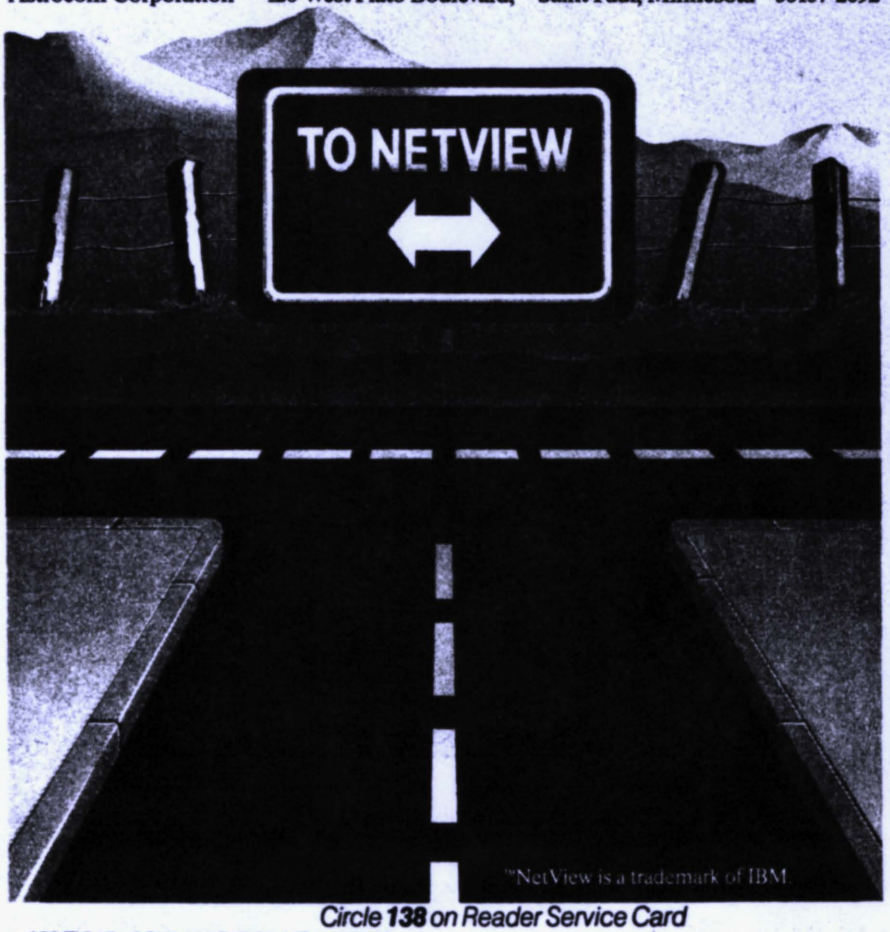
For the smooth and straight road to LPDA-2, call: (612) 227-8651



astrocom

Your Link To Better Communications

Astrocom Corporation 120 West Plato Boulevard, Saint Paul, Minnesota 55107-2092



™NetView is a trademark of IBM

Circle 138 on Reader Service Card

Streams

approach is simplicity and uniformity: A programmer uses one approach to network programming, regardless of the type of network.

In essence, Streams allows a programmer to see a toolbox of capabilities: TLI tools, for example, make it possible to set up connections easily. The fact that the connection is over TCP/IP, OSI, DECnet, or Novell SPX is irrelevant to the application programmer.

TLI shields the programmer from the intricacies of transport providers. Upper-layer providers, such as RPCs, can be put into this framework with ease. Netwise, for example, uses Streams and TLI to tie its RPC technology into Novell's NetWare and other network architectures.

With Streams, network programmers can concentrate on their work without worrying about the lower layers of the network.

RPC tools provide an even higher level of functionality, allowing the programmer to treat the remote computer as a simple extension of local memory. Procedures are called, parameters passed, and values returned.

Both RPC and TLI are technologies that make network programming easier. Streams is an underlying structure that works beneath both. Streams allows standards such as TLI to be easily developed and implemented. Standards and tools such as TLI and RPCs give the programmer a high degree of portability, flexibility, and power. The result is that the user can begin treating the network less like a monolithic whole and more like a set of building blocks, in which the best module for the job at hand is simply dropped into place. ■

Carl Malamud is a contributing editor to DATA COMMUNICATIONS. This article is an excerpt from his forthcoming book *Analyzing Novell Networks* (Van Nostrand Reinhold, 1990).

The author wishes to thank Art Sabsevitz of AT&T Bell Laboratories for his review and comments.